THE RUNTIME DIFFERENCE BETWEEN THE A* AND DIJKSTRA'S PATHFINDING ALGORITHMS IN SOLVING MAZE PROBLEMS

Computer Science Extended Essay

Research Question

What is the difference between the runtime efficiency of Dijkstra's and the A* pathfinding algorithms in finding the shortest path in mazes with varying size?

Candidate Code: hcg315

CS EE World
https://cseeworld.wixsite.com/home
May 2021
26/34
B
Submitter Info: Anonymous

Word Count: 3988

Table of Contents

1. Introduction

Research Question: What is the difference between the runtime efficiency of Dijkstra's and the A* pathfinding algorithms in finding the shortest path in mazes with varying size?

Pathfinding algorithms (finding the shortest path between two set points on a grid), although might sound related only to technology, are an integral part of life. We, humans, have to determine our path in tasks like commuting to work, assessing the length and other factors of the road. Computers, however, need algorithms to determine the shortest path in such problems (Krafft 1,2). Pathfinding in computers is used in "navigation, video games, robotics, logistics" and others. (Algfoor, Sunar and Kolivand 1-3)

There are different pathfinding algorithms, from which Dijkstra's (Khan) and the A* algorithm (Mehta et al.) stand out as one of the most used algorithms.

This extended essay aims to investigate the runtime difference between Dijkstra's and the A* pathfinding algorithm in finding the shortest path from a starting and ending point in a maze problem with multiple paths between the starting and ending points in the maze.

This paper can aid especially in the video game and navigation fields. In real time strategy games such as Age of Empires, numerous units (armies, workers, etc.) constantly pathfind around in a large map consisting of a 256x256 grid (Cui and Shi 128,129). Even though the game is able to compute the paths of the units without visible lag, the players have consistently complained about units getting stuck or traverse a nonsensical path (H. Patel). Increasing the efficiency of the pathfinding algorithm used can aid in the better playability of the game by solving the existing problems.

Furthermore, autonomous drones are also starting to be a part of our lives, potentially shipping crucial cargo in the future. Employing the most efficient pathfinding algorithm can help in reducing costs and increasing mission success chances of such drones (Fu et al. 1,2). To investigate the difference between the algorithms, a maze generation algorithm (Recursive backtracking with dead end culling) along with the A* and Dijkstra's algorithms were programmed in C#. Their runtimes on different sizes of procedurally generated mazes were measured and analyzed.

2. Background Information

2.1 Procedural Maze Generation

Mazes are so old as to inspire Greek myths like the Minotaur and the labyrinth and are used currently as entertainment in means of video games (Pac-man, many roguelike games, etc.) or simply as puzzles to solve on the backs of newspapers (Hybesis). With the use of computers, completely random and very large mazes can be generated. There are many different procedural maze generation algorithms. (Pullen)

2.1.1 Maze Properties

Mazes have many different properties, indicating their nature. The properties relevant to this investigation are shown below.

Perfect mazes are defined by three properties: not having any passage loops, not having any isolated nodes and having only one path between any node pair in the maze. There are numerous ways to generate and solve such mazes as they are the most commonly used maze type. (Foltin 7)

Braided mazes, unlike perfect mazes, have no dead ends and may have multiple paths of varying length between two node in the maze (Foltin 7). Although there are different ways to generate such mazes (Ioannidis 31-35), the algorithms are much rarer since this maze type isn’t as popular as perfect mazes.

Partial braided mazes are a combination of dead ends and loops. The ratio between the dead ends and loops can be calculated or manipulated. Similar to braided mazes, algorithms for the procedural generation for partial braided mazes are uncommon.
(Pullen)

The **elitism** of a maze is how much the solution of the maze covers its area. An elitist maze has a shorter and more direct solution, a non-elitist maze has a longer solution, covering more of its area. If there are multiple solutions, the elitism applies to the shortest path.
(Pullen)

2.1.2 The Recursive Backtracker Algorithm

A simple way to generate perfect mazes is the 'Recursive Backtracker' algorithm, which is based on the 'depth first search technique' (DFS). "The DFS algorithm wanders through the graph in a depth-oriented way". (Foltin 20-22) The algorithm travels whenever possible to a neighbor of the current node, and if it can't, it goes back to the previous vertex until it has iterated through all vertices. While generating a maze, a grid with node which all have 4 walls around them is firstly created. Then when the algorithm is traveling between vertices (or nodes), the wall between the two are destroyed, eventually generating a maze by boring walls though the grid. (Hybesis)

The algorithm has two possible implementations, either by recursion or iterative. (Ioannidis 23-25) The recursive version uses a lot of memory and is prone to overflow errors, while the iterative version uses a stack to store less data.

The steps for the iterative implementation and an illustration for the generation (figure 1) and a sample (figure 2) can be seen below.

1. Choose a starting point in the field.
2. Randomly choose a wall at that point and carve a passage through to the adjacent node, but only if the adjacent node has not been visited yet. This becomes the new current node.
3. If all adjacent node have been visited, back up to the last node that has uncarved walls (shown by the yellow points in figure 1) and repeat step 2.
4. The algorithm ends when the process has backed all the way up to the starting point. (Buck Maze Generation: Recursive Backtracking)

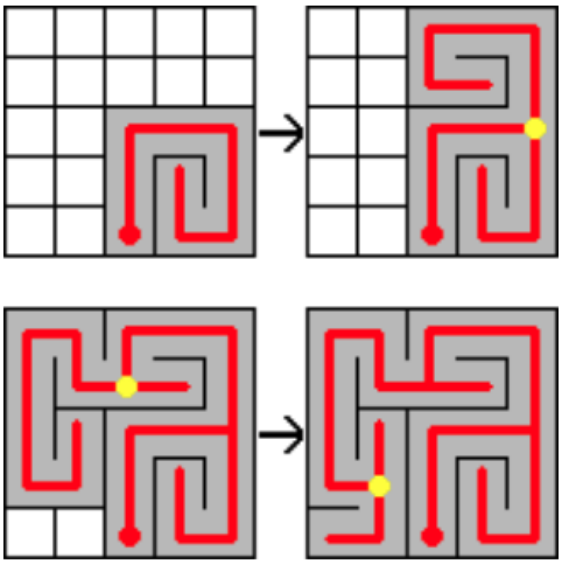


Figure 1: Image depicting recursive backtracker steps (Foltin 22)

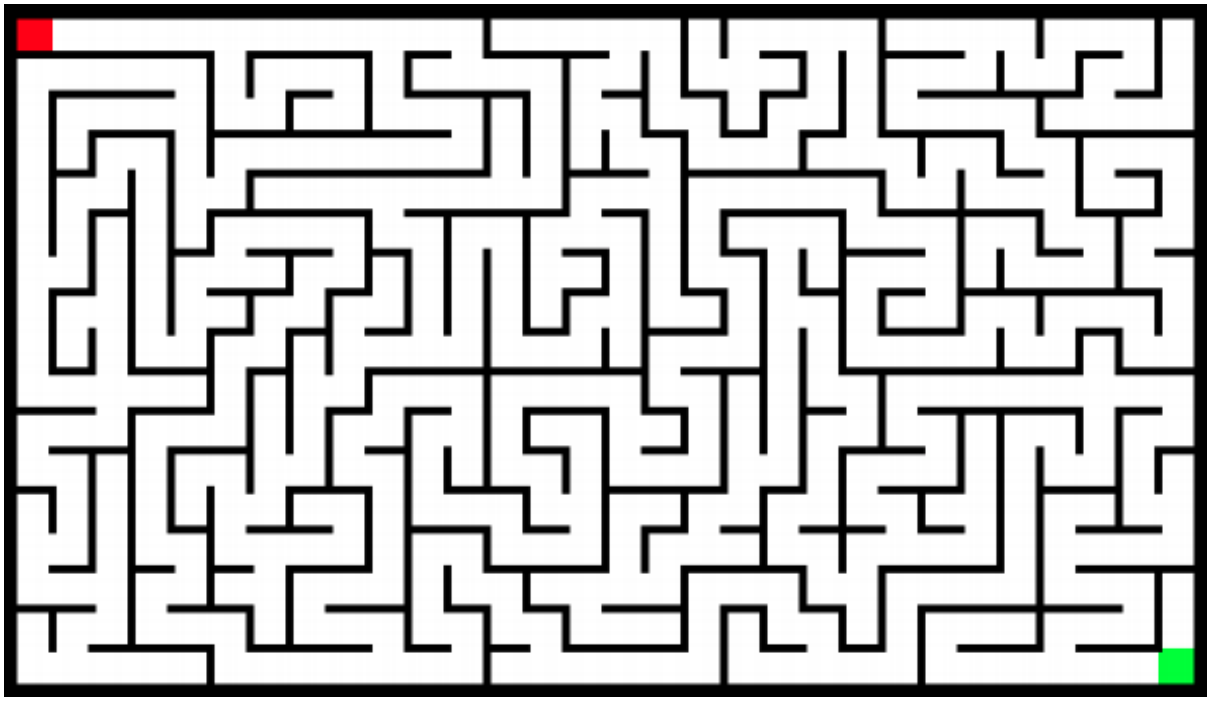


Figure 2: Image depicting a maze created using the recursive backtracking algorithm (Ioannidis 28)

2.1.3 Dead End Culling

While a vast number of algorithms exist for perfect maze generation, that is not the case for braided maze generation. Even though algorithms such as “Random Restarts” (Ioannidis 35-39) exist, they are uncommon. An easy way to obtain braided mazes is applying “dead-end culling” to a perfect maze, changing the walls on the dead ends so that they no longer are dead ends. Dead end culling also provides the option for exceptionally easy partial braiding (with desired dead end to loop ratios). Pseudocode for dead end culling can be seen below.

1. Iterate through all node
2. If current node is a dead end (3 walls including outside borders) remove random wall excluding outside borders

(Buck Mazes for Programmers)

2.2 Pathfinding Algorithms

Pathfinding algorithms are aimed to find the shortest possible path between two set points. It has many applications such as street navigation in Google Maps, video games and maze solving. There is a multitude of pathfinding algorithms. (Algfoor, Sunar and Kolivand 1-3)

2.2.1 Dijkstra's Pathfinding Algorithm

Dijkstra's algorithm expands outwards from its starting point until it meets the ending point. There is a 100 % chance that the algorithm will find a shortest path (there can be multiple shortest paths, with the same length). The illustration below shows the algorithm working on a blank grid. The blue nodes have been visited by the algorithm, and the pink and purple nodes are the start and end points respectively. (A. Patel)

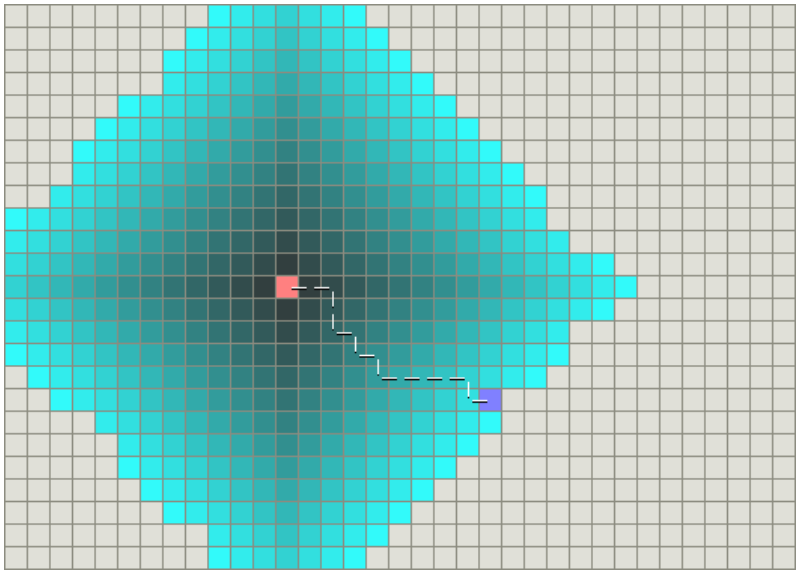


Figure 3: Image depicting Dijkstra's algorithm (A. Patel)

Pseudocode for the algorithm can be seen below.

```
// Dijkstra's Algorithm

// Set each node's position to infinity
for each node in the graph
    set the node's distance to infinity
    set the node's parent to none

// Create an unexplored set
let the unexploredSet equal a set of all the nodes

while the unexploredSet is not empty

    // Get the current node
    let the currentNode equal the node with the smallest distance
    remove the currentNode from the unexploredSet

    // Check completed
    if currentNode's position is your goal
        Congratz! You've found the end! Backtrack to get path

    // Get all the neighbors
    for each neighbor (still in unexploredSet) to the currentNode

        // Calculate the new distance
        let newDist equal currentNode's dist plus distance between
            the currentNode and the neighbor

        // Check to see if the new distance is better
        if newDist is less than currentNode's distance
            set neighbor's distance to newDist
            set neighbor's parent to currentNode
```

Figure 4: Pseudocode for Dijkstra's algorithm (Swift Easy Dijkstra's Pathfinding)

2.2.2 The A* Pathfinding Algorithm

A* is the most popular choice for pathfinding in video games (Mehta et al.) among others, and is a modification of Dijkstra's algorithm, and expands in the direction towards the goal. It uses a heuristic function (finding an approximate solution) to find out paths which seem to be leading to the goal and also favors paths which have the shortest path from the starting point. It always finds a shortest path. (A. Patel)

Calculating the cost of a node

The two goals of the algorithm (distance to the start and end nodes) are weighted by the f-cost, which is the overall ‘cost’ of a node based on its distance to the start node (g-cost) and the projected distance to the end node (h-cost). The f-, g- and h-costs are explained below.

f-cost: total cost of the node (g-cost + h-cost)

g-cost: length of the path between the node and the start

h-cost: heuristic, distance estimated to be between the node and the end. It can be acquired by using the Pythagorean theorem on the x- and y-difference between the end and current node, although other methods exist (Peters)

The pseudocode can be seen below.

```
// A* (star) Pathfinding

// Initialize both open and closed list
let the openList equal empty list of nodes
let the closedList equal empty list of nodes

// Add the start node
put the startNode on the openList (leave it's f at zero)

// Loop until you find the end
while the openList is not empty

    // Get the current node
    let the currentNode equal the node with the least f value
    remove the currentNode from the openList
    add the currentNode to the closedList

    // Found the goal
    if currentNode is the goal
        Congratz! You've found the end! Backtrack to get path

    // Generate children
    let the children of the currentNode equal the adjacent nodes

    for each child in the children

        // Child is on the closedList
        if child is in the closedList
            continue to beginning of for loop

        // Create the f, g, and h values
        child.g = currentNode.g + distance between child and current
        child.h = distance from child to end
        child.f = child.g + child.h

        // Child is already in openList
        if child.position is in the openList's nodes positions
            if the child.g is higher than the openList node's g
                continue to beginning of for loop

        // Add the child to the openList
        add the child to the openList
```

Figure 5: Pseudocode for the A* algorithm (Swift Easy A*)

Example illustrations of the algorithm can be seen in figures 6 and 7.

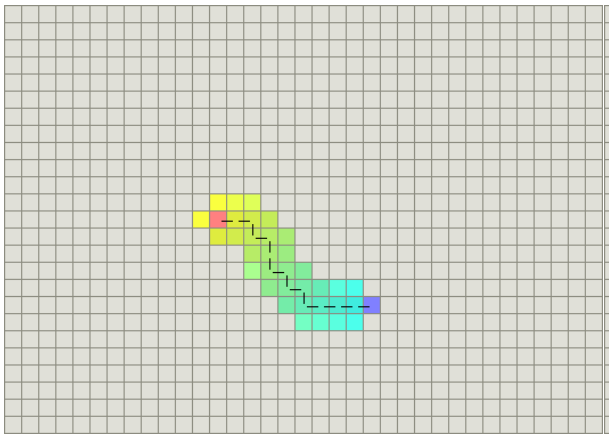


Figure 6: Unobstructed A* algorithm (A. Patel)

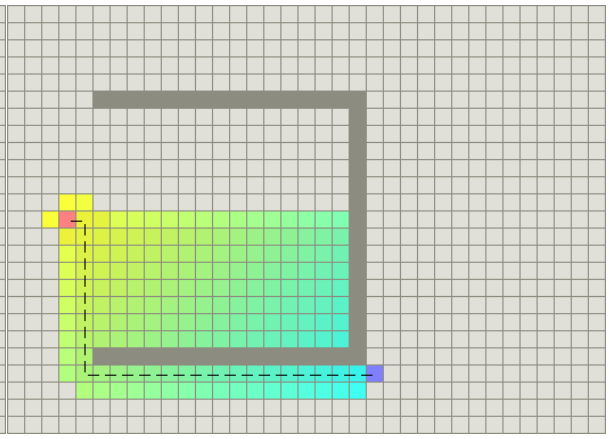


Figure 7: Obstructed A* algorithm (A. Patel)

2.3 Time Complexity of Algorithms

Big O notation is commonly used for the time complexity (the relation of runtime as the input gets larger). Big O notation has different cases for the best, average and worst outcomes, the worst outcome being the most used. (Cormen et al. 43-50)

Worst-Case

The worst-case complexity is done most frequently since it is easy to calculate and can show a general picture. Although it is useful, it might be too pessimistic in some cases or ignore the complete picture. (Chauan)

Best-Case

The best case shows the lower bound of the time taken for the algorithm. This isn't popular to analyze since it can't provide reliable information. An algorithm iterating over a very large data set could have small best-case time complexity, while needing years to finish operating on average. (Chauan)

Average-Case

The average case shows the time complexity of the algorithm in a more realistic and whole sense than both the worst- and best- case. It is however difficult to calculate since the 'average set of inputs' is needed to be known. The set of inputs are assessed by their probability and how much time they take, calculating the expected value. This nature of input is then used to get the time complexity. (Zeil)

2.3.1 Time complexity of Dijkstra's Algorithm

The worst-case time complexity of Dijkstra's Algorithm is $O(V^2)$, with V being the amount of vertices (nodes) in the graph. ("Shortest Path Algorithms")

To get the average case, the expected value of iterations is needed, which is wholly dependent on the input. The nature of the input is needed to be known to find the average case of Dijkstra's algorithm.
(Nilsson)

2.3.2 Time complexity of the A* algorithm

The worst-case time complexity of the A* algorithm is the same as Dijkstra's. This is due to both algorithms having to iterate through all the nodes in the worst-case, resulting in the same amount of iterations. This is also the case for the best-case, as a direct path towards the end node without any diverging paths would result in the same amount of iterations as well. However, due to using a heuristic function, the average case time complexity is aimed to be improved. (Bast)

Due to the usage of a heuristic function and the required nature of input, the average time complexity can only be determined by finding the 'quality' of the heuristic and nature of the input. (Russell and Norvig 97-104)

3. Hypothesis

Even though the best- and worst-case time complexities of Dijkstra's algorithm and A*, a heuristic function is used to make the A* algorithm be more guided towards the goal to decrease the amount of iterations, hence decreasing the overall runtime. Therefore, the A* algorithm is expected to have a shorter runtime than Dijkstra's on average.
As the size of the maze gets smaller, the cost of the heuristic function is expected to get more significant, which will result in the difference between the runtimes of the two functions getting smaller.

The average time complexities of the two algorithms can't be used to predict their runtimes because they cannot be determined without running tests on the algorithms.

4. Methodology

The recursive backtracker, dead end culling; Dijkstra's and the A* pathfinding algorithms were written for the investigation (see appendix 1). The code was written with the steps in the background information. The IDE "Visual Studio" was used for the C# implementation for the algorithms in the investigation. Although diagonal movement isn't allowed, the Pythagorean theorem is still used to calculate the heuristics function (h-cost) for the A* algorithm to maintain its integrity from real life applications.

The runtime is measured by the “System.Diagnostics.Stopwatch” class, which is the class commonly used for measuring runtime. (Allen)

The startup runtime output shows the speed at which the computer is running at the time of startup. It accesses and edits an integer variable 100000000 times.

The two pathfinding algorithms are to find the shortest path from the starting point to the ending point (declared to be at contrasting fifths of the whole grid. For example, on a 100x100 grid, the starting point is at coordinates relative to the top left corner (20,20), and the ending point at (80,80)). This allows the algorithms to venture behind the starting and ending points, rather than them being on opposite ends of the grid, not allowing any movement back.

The output of the code will be easy to transfer manually to MS Excel for analysis. Sample code output for a single trial is below.

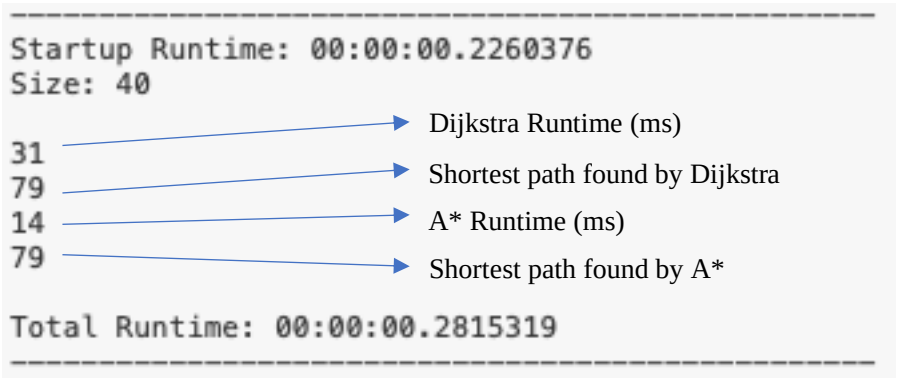


Figure 8: Sample code output for a single trial

4.1 Controlled Variables:

Maze characteristics: The characteristics of the maze are to be kept identical throughout the whole tests. Three aspects of mazes are listed below.

Elitism: Constraining the shortest path length between the start and end points could help in reducing random errors for the end result. Doing this however would be very tedious and there is no clear algorithm which improves upon the elitism.

Braiding density: The maze will not be partially braided, since randomly picking dead ends which won't be removed will increase the effect of random errors on the end result. As random errors due to the unchangeable elitism of the maze will be caused, it was opted out of having partial braiding.

Tile costs/weights: Having random tile weights (the path length between two node) would increase random errors just like the partial braiding, and therefore not implemented.

The computer that will be used (Macbook Air 2017) has the following specifications:

- 1.8GHz dual-core Intel Core i5 processor with 3MB shared L3 cache
- 8GB of 1,600MHz LPDDR3 RAM
- Intel HD Graphics 6000 (Haslam)

The exact same code except for the size of the maze (the independent variable) will be used throughout the tests.

The IDE Visual Basic and language C# will be used for the code implementation throughout the tests.

4.2 Procedure Steps

1. Mazes of sizes ranging from 40x40 to 320x320 with intervals of 40 (40x40, 80x80, etc.) with 10 repeats for each are generated by the recursive backtracking algorithm.
2. The dead-end culling algorithm is used to turn the perfect mazes generated in step 1 to braided mazes.
3. The mazes are solved by Dijkstra's pathfinding algorithm and the A* algorithm. The runtimes and shortest path lengths for each maze are recorded.

5. Data Presentation

After the tests, the output of the code (appendix 2) was manually translated into MS Excel, where the average and uncertainty of all the trials were calculated (raw data tables in appendix 3) then formed into the following tables.

Maze Size Against Average Shortest Path Found

The two algorithms aren't separated in this table since they had the exact same output.

Maze Size	Shortest Path
40x40	78 ± 20
80x80	148 ± 15
120x120	219 ± 19
160x160	287 ± 21
200x200	367 ± 33
240x240	426 ± 35
280x280	494 ± 25
320x320	559 ± 45

Table 1: Maze size against average shortest path

Maze Size Against Average Runtime of Dijkstra's and the A* Algorithm

Maze Size	Dijkstra Runtime (ms)	A* Runtime (ms)
40x40	26,6 ± 4,0	10,3 ± 8,0
80x80	417,9 ± 19,5	134 ± 58,0
120x120	2131,8 ± 60,5	802 ± 317,0
160x160	7780,9 ± 827,5	2706,9 ± 762,0
200x200	23049,9 ± 2226,5	9733,8 ± 4736,5
240x240	51978,3 ± 3651,0	24833,6 ± 13527,0
280x280	109061,1 ± 29995,0	65271,1 ± 19982,5
320x320	184742,9 ± 20941,0	121041,2 ± 57608,0

Table 2: Maze size against average runtimes of Dijkstra and A*

Maze Size Against Ratio Between the Average Runtimes of Dijkstra and A*

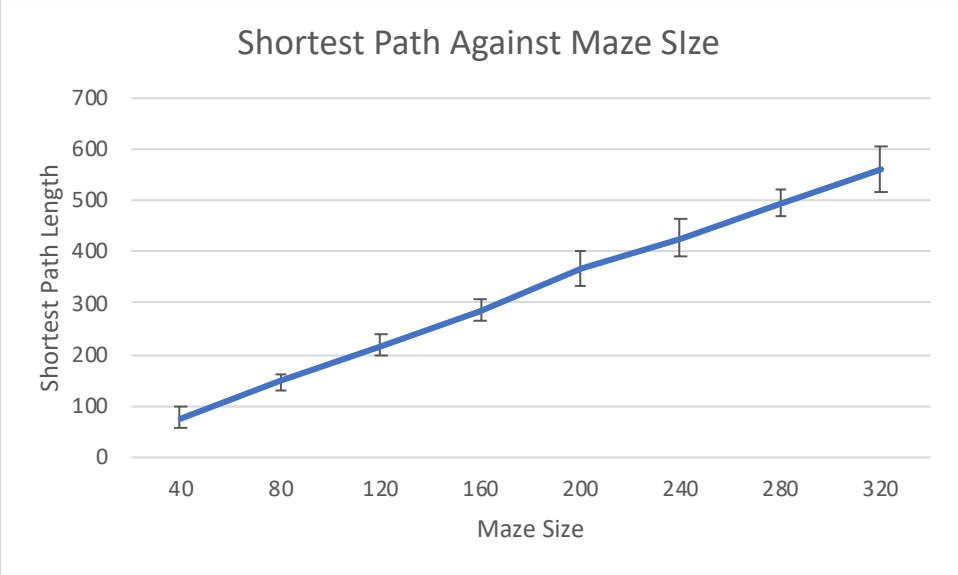
The values were calculated simply by doing the operation Dijkstra Runtime divided by A* Runtime.

Maze Size	Ratio
40	2,58
80	3,12
120	2,69
160	2,87
200	2,37
240	2,10
280	1,67
320	1,53

Table 3: Maze size against ratio between average runtimes of Dijkstra and A*

The tables 1, 2 and 3 were used to create the following graphs (graphs 1, 2, 3, 4, 5).

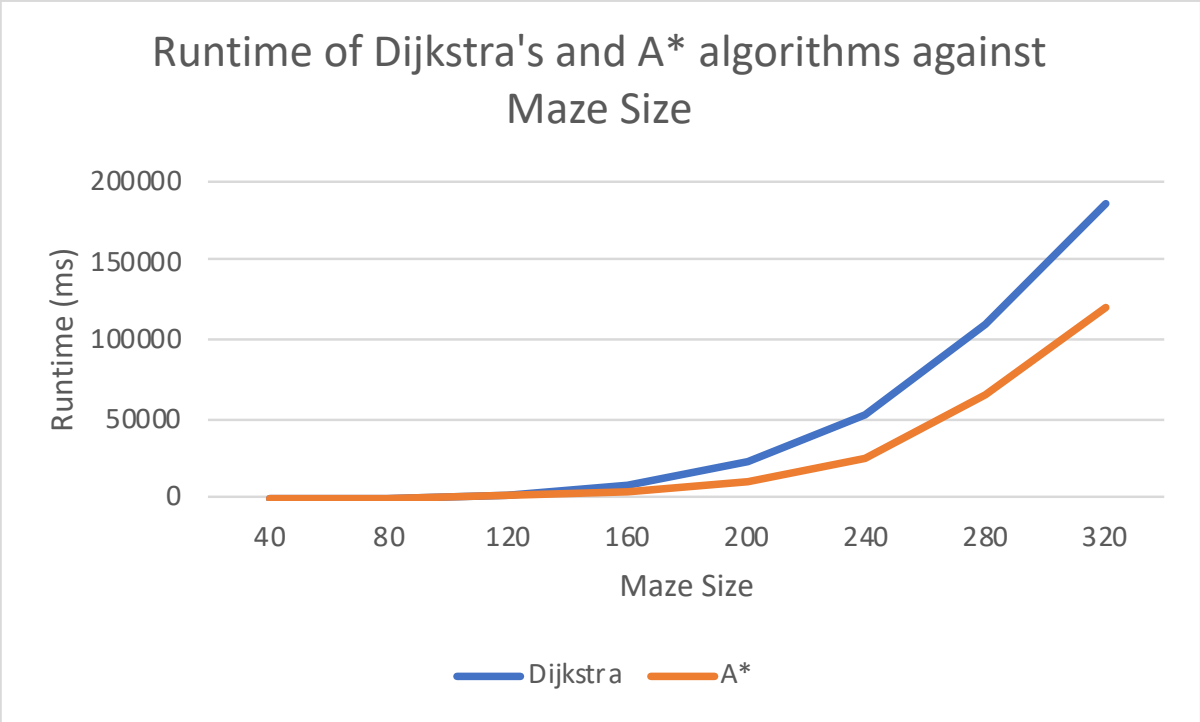
Shortest Path against Maze Size



Graph 1: Maze size against shortest path

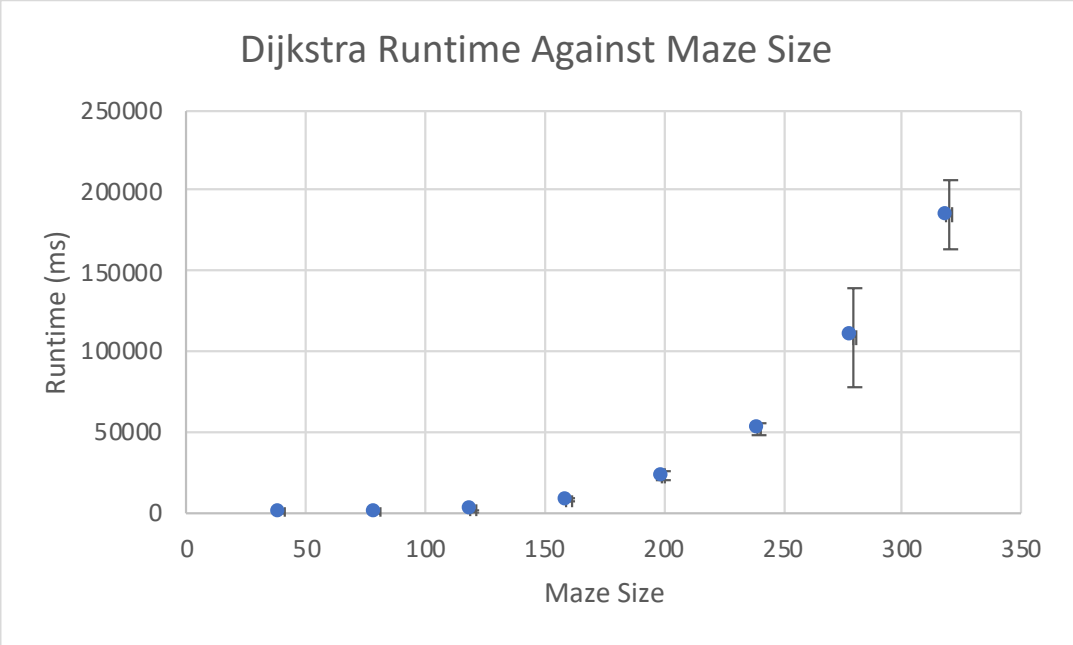
Runtime of Dijkstra's and the A* Pathfinding Algorithms against Maze Size

Uncertainties were not added to this graph as they would obstruct the view.



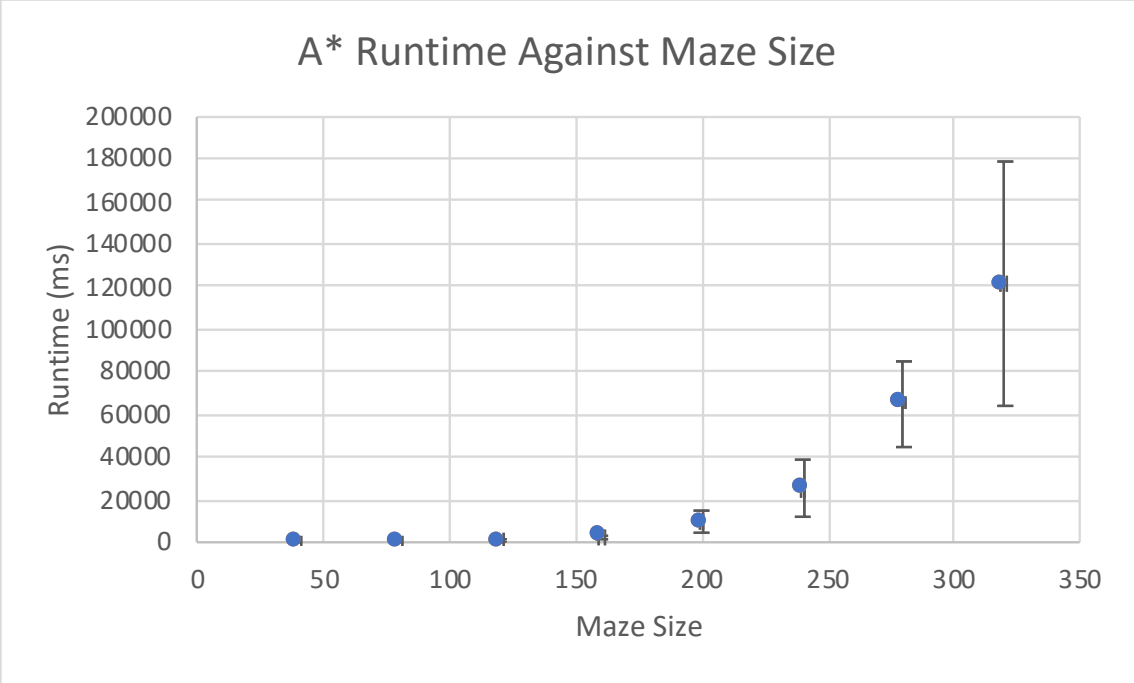
Graph 2: Maze size against Dijkstra and A* runtime

Runtime of Dijkstra's Algorithm against Maze Size



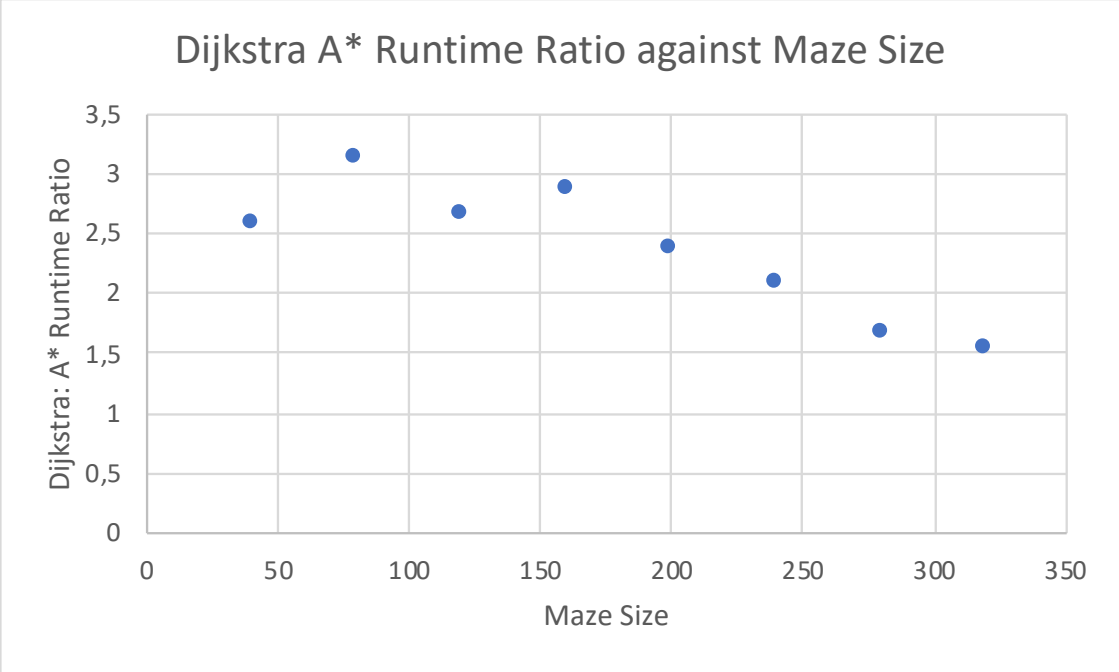
Graph 3: Maze size against Dijkstra runtime

Runtime of the A* Algorithm against Maze Size



Graph 4: Maze size against A* runtime

Dijkstra to A* Runtime Ratio against Maze Size



Graph 5: Maze size against Dijkstra to A* ratio

The graphs 1, 2, 3, 4, 5 were against the length of one axis of the maze (taken as “Maze Size”) instead of its area (total number of nodes in the maze) because the shortest path length is linear to it value unlike its area. This helps in analyze the runtimes of the algorithms because they operate mainly on the shortest path and not the whole maze.

6. Data Analysis

The shortest paths found for the same mazes by the two algorithms were always the same, showing that both are capable of finding the shortest path successfully, which was predicted in the hypothesis. Seeing graph 1, the relationship between the size of one side of the maze is linear to the shortest path between the specified starting and ending points.

Seeing graph 2 and table 2, the runtime of the A* algorithm is always shorter than Dijkstra’s.

It can be understood from graphs 3 and 4 that the curves roughly have the same shape (exponential increase), which relates to the fact that they derive from the same algorithm.

Seeing graph 3, Dijkstra's algorithm has unstable error bars, even while tending to have a bigger uncertainty as the maze gets larger. This is however not a direct correlation as the uncertainty for the 280x280 maze (± 29995) is larger than the uncertainty for the 320x320 maze (± 20941). These two uncertainties are also very large, showing that there are random errors in the form of the complexities of the mazes.

Seeing graph 4, the A* algorithm has its error bars widen steadily in correlation with the runtime, but the uncertainties are very large, especially at points 40 (nearly as large as the runtime value), 240 and 320 (approximately half the value). This shows that there have been a very large range of random errors throughout the tests. Despite this range of runtime, a very little portion of it falls in the range of the runtime of Dijkstra's algorithm, showing that A* has less runtime. This large range can be attributed to the predicting characteristics of the heuristic function and the varying shortest solution of the maze even when size is constant.

Finally, from the last graph it can be seen that there is a general decrease of the ratio between the runtimes of the two algorithms. This means that the difference between the two gets smaller as the size of the maze increases. This contradicts the hypothesis, as it was predicted that the difference would decrease as the size of the maze gets smaller. This change in difference might be due to the heuristic function not being able to approximate the h-cost as well in larger mazes.

From the data, it can be seen that even though their difference decreases as the maze gets larger, the A* star algorithm performs much faster that Dijkstra's algorithm. In smaller mazes such as in an 80x80 grid, it can find the same shortest path approximately 3 times faster, doing the same operation on a 320x320 grid 1,5 times faster than Dijkstra's algorithm.

7. Limitations

One of the major limitations of the methodology as seen in the data analysis is the fact that the runtime values of both algorithms are very imprecise. This is mainly caused by the fact that different mazes of the same size can have different complexities or difficulty, also having shortest paths with different lengths. This was also talked about in the control variables section, where it was stated that it was very hard to procedurally generate mazes with very similar difficulty and similar shortest path length (similar elitism). The impreciseness of the result degrades its reliability.

Another limitation which contributed to the impreciseness of the result is the fact that only 10 trials are being done for each selected maze size. This renders many different mazes untouchable and increases random errors vastly as the mazes are randomly generated. It also doesn't show the worst- and best-case scenarios.

The lack of variation of the dead ends and loops (the maze not being partially braided) might affect the results in a real-life application side. Although having random decisions between dead ends and loops would significantly increase the random errors which are already very high, they would represent a maze in a game or actual city streets much better than the current fully braided model. The same can be said about putting weights into paths between nodes.

The last graph showed that the ratio between the algorithms was decreasing but didn't show a 1:1 ratio. It cannot be known if A* will still be superior if gone into larger mazes.

8. Further Development

To increase precision, the independent variable could be changed to a specific shortest path length in a specific maze size. This could be done through setting a wanted path length and iterating until a maze which has the wanted shortest path length has been found and doing the usual tests on it. This could reduce random errors, even though the code would take significantly more time to operate.

Instead of looking at 10 samples from a specific maze size, all the mazes in that size can be evaluated. Algorithms which are uniform (can create all possible mazes) such as Wilson's or the Aldous-Broder algorithm can be used for the maze generation (Pullen). This would include the best- and worst-case scenarios in the result. This would also make finding the precise average time complexity of both algorithms possible as the nature of the input can be analyzed wholly.

Partially braiding the maze (as opposed to full braiding) or assigning weights for paths between nodes can make the maze resemble a web of streets or a videogame map more(with dead-ends, loops and harder paths to traverse), even though it would decrease the precision drastically.

9. Final Conclusion

This investigation aimed at finding the runtime difference between Dijkstra's and the A* pathfinding algorithms at solving maze problems with varying sizes. After the experiment

and result analysis, it can be concluded that the A* algorithm performs much faster than Dijkstra's algorithm.

This investigation aimed at finding the runtime difference between Dijkstra's and the A* pathfinding algorithms at solving maze problems. After doing the experiment by running the two algorithms on procedurally generated mazes of varying sizes and recording the runtimes, it can be seen that the runtime of both algorithms increased exponentially as the size (hence the shortest path) of the mazes increased. It was observed that the uncertainties of the Dijkstra algorithm increased along with the increase of the average runtime, albeit without a clear correlation. Similarly, the uncertainty of the A* algorithm increased with the average runtime, but it was consistent and much larger than Dijkstra's. The runtime of the A* algorithm was always better than Dijkstra's, with the difference between them reducing from 3 times to 1,5 times as the maze size increased.

The problem of the mazes having different lengths of shortest paths even when the size is the same (resulting in large random errors) can be solved by having the independent variable as shortest path length instead of size. Also, partially braiding the maze might lead to more realistic results or assessing all mazes for a single size can increase precision and aid in acquiring the time complexity.

Since the A* algorithm was found to be faster than Dijkstra's algorithm, it is advised to use the A* algorithm in pathfinding problems which resemble or are non-weighted mazes with multiple paths going from the start to the goal to decrease the runtime required.

10. Bibliography

Algfoor, Zeyad Abd, Mohd Shahrizal Sunar, and Hoshang Kolivand. "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games." *International Journal of Computer Games Technology,* vol. 2015, 2015, pp. 13. *Hindawi,* www.hindawi.com/journals/ijcgt/2015/736138 /. Accessed 20 November 2020.

Allen,Sam. "C# Stopwatch Examples." *Dotnetperls,* 2020, https://www.dotnetperls.com/stopwatch. Accessed 20 November 2020.

Bast,Hannah. "A*, Landmarks, Dijkstra." *Efficient Route Planning,* 9 May 2012, University of Freiburg, Freiburg, ad-teaching.informatik.uni-freiburg.de/route-plann ing-ss2012/lecture-3.mp4. Accessed 20 November 2020.

Buck,Jamis. "Maze Generation: Recursive Backtracking." *The Buckblog,* 2010, weblog.jamisbuck.org/2010/12/27/maze-genera tion-recursive-backtracking.html. Accessed 01 October 2020.

Buck,Jamis. *Mazes for Programmers: Code Your Own Twisty Little Passages.* Edited by Jacquelyn, Carter, Pragmatic Bookshelf, 2015.

Chauan,Sankalp., creator, *Analysis of Algorithms | Set 2 (Worst, Average and Best Cases) | GeeksforGeeks.* Youtube, Uploaded by Geeksforgeeks, 18 February 2019, www.youtube.com/watch?v=rlzpz8es_6k&feature=emb_lo go. Accessed 20 November 2020.

Cormen, Thomas H., Leiserson Charles E., Rivest Ronald L., and Stein Clifford. *Introduction to Algorithms.* 3rd ed., The MIT Press, 2009, Pp. 43-50, 682-683.

Cui,Xiao, and Hao Shi. "A*-based Pathfinding in Modern Computer Games." *IJCSNS International Journal of Computer Science and Network Security,* vol. 11, no. 1, 2011, P. 128,129. *Researchgate,* www.researchgate.net/publication/267809499 _a-based_pathfinding_in_modern_computer_games. Accessed 20 November 2020.

Foltin,Martin. *Automated Maze Generation and Human Interaction.* Masaryk University Faculty of Informatics, 2011, pp. 7-9, 20-22. is.muni.cz/th/xofma/thesis.pdf. Accessed 1 October 2020.

Fu,Zhangjie, Jignan Yu, Guowu Xie, Yiming Chen, and Yuanhang Mao. "A Heuristic Evolutionary Algorithm of UAV Path Planning." *Wireless Communications and Mobile Computing,* vol. 2018, 2018, P. 1,2. *Hindawi,* www.hindawi.com/journals/wcmc/2018/2851964 /#copyright. Accessed 20 November 2020.

Haslam,Karen. "Apple MacBook Air (2017) review." *Macworld,* 2018, https://www.macworld.co.uk/review/macbook-air-2017 -3659879/. Accessed 20 November 2020.

Hybesis, -. H.urna. "Maze generations: Algorithms and Visualizations." *Medium,* 2019, medium.com/analytics-vidhya/maze-generatio ns-algorithms-and-visualizations-9f5e88a3ae37. Accessed 20 November 2020.

Ioannidis, Petros L. *Procedural Maze Generation.* National and Kapodastrian University of Athens, 2016, pp. 23-25, 28, 31-39. pergamos.lib.uoa.gr/uoa/dl/frontend/file/lib/default/data/1324569/theFile/1324570. Accessed 01 October 2020.

Khan, Zafer Ali. *Comparison of Dijkstra's Algorithm with other proposed algorithms.* 2016. Virtual University of Pakistan. Researchgate, https://www.researchgate.net/publication/309771211_Comparison_of_Dijkstra's_Algorithm_with_other_proposed_algorithms

Krafft,Carina. *Implementation and Comparison of Pathfinding Algorithms in a Dynamic 3D Space.* University of Applied Sciences Hamburg Faculty of Design. Media and Information Department Media Technology, 2019, p. 1,2. users.informatik.haw-hamburg.de/~schumann/BachelorArbeitCarinaKrafft.pdf. Accessed 20 November 2020.

Mehta,Parth, Hetasha Shah, Soumya Shukla, and Saurav Verma. "A Review on Algorithms for Pathfinding in Computer Games." *IEEE Sponsored 2nd International Conference on Innovations in Information Embedded and Communication Systems ICIIECS'15,* Karpagam College of Engineering, Tamil Nadu, 19 March 2015.

Patel,Amit. "Introduction to A*." *Stanford,* 2020, theory.stanford.edu/~amitp/gameprogramming/astarcomparison.html#:~:text=a*%20is%20the%20most%20popular,a%20heuristic%20to%20guide%20itself. Accessed 01 October 2020.

Patel, Hardik. “How does Age of Empires II pathfinding algorithm work?”. *Quora,* https://www.quora.com/How-does-Age-of-Empires-II-pathfinding-algorithm-work. Accessed 20 November 2020.

Peters, Mark. “Re: What are some good methods to finding a heuristic for the A* algorithm?”. *Stack Overflow.* https://stackoverflow.com/questions/5687882/what-are-some-good-methods-to-finding-a-heuristic-for-the-a-algorithm. Accessed 20 November 2020.

Pullen, Walter D. "Maze Classification." *Astrolog,* 2015, www.astrolog.org/labyrnth/algrithm.htm#perfect. Accessed 01 October 2020.

"Shortest Path Algorithms." *Hackerearth,* 2020, https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/. Accessed 20 November 2020. Nilsson,Stefan. "How to analyze time complexity: Count your steps." *Yourbasic,* 2020, https://yourbasic.org/algorithms/time-complexity-explained/. Accessed 20 November 2020. Russell, Stuart J., and Peter Norvig. *Artificial Intelligence a Modern Approach.* Edited by Mona Pompili, Prentice-Hall, 1995, Pp. 97-104.

Swift,Nicholas. "Easy A* (star) Pathfinding." *Medium,* 2017, medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2. Accessed 01 October 2020.

Swift,Nicholas. "Easy Dijkstra’s Pathfinding." *Medium,* 2017, medium.com/@nicholas.w.swift/easy-dijkstra s-pathfinding-324a51eeb0f. Accessed 01 October 2020.

Zeil, Steven J. "Analysis of Algorithms: Average Case Analysis." *Old Dominion University,* 2017, https://www.cs.odu.edu/~zeil/cs361/f17/public/averagecase/index.html. Accessed 20 November 2020.

11. Appendix

11.1 Body of Code

It must be noted that the code written for the investigation isn't documented.

```
using System;
using System.Collections;
using System.Collections.Generic;

public class Node
{
    public int x;
    public int y;
    public Node(int xPosition, int yPosition) //argument for x,y positions in Setup()
    {
        x = xPosition;
        y = yPosition;
    }
    public bool isVisited = false;
    public bool[] walls = new bool[4];

    //Djkstra components
    public int dDistance = int.MaxValue;
    public Node previous = null;

    //A* components
    public int fCost;
    public int gCost;
    public int hCost;
}

namespace General_Test
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("------------------------------------------------");
            var mainWatch = System.Diagnostics.Stopwatch.StartNew();
            //startup
            int iterate = 5;
            var Watch = System.Diagnostics.Stopwatch.StartNew();
            for (int i = 0; i < 100000000; i++)
            {
                iterate++;
            }
            Watch.Stop();
            Console.Write("Startup Runtime: ");
            Console.WriteLine(Watch.Elapsed);
```

```
        //important variables
        int size = 40;
        bool braidOn = true;
        bool captionsOn = false;
        int trialAmount = 1;

        Console.Write("Size: ");
        Console.WriteLine(size);
        Console.WriteLine("");

        for (int globalCounter = 0; globalCounter < trialAmount; globalCounter++) //repeat
test
        {
            Stack stack = new Stack();
            Node[,] allNodes = new Node[size, size];


            List<Node> FindUnvisitedNeighbors(Node inputNode)
            {
                List<Node> neighbors = new List<Node>();
                int x = inputNode.x;
                int y = inputNode.y;

                if (x != 0)
                {
                    if (!allNodes[y, x - 1].isVisited)
                    {
                        neighbors.Add(allNodes[y, x - 1]);
                    }
                }
                if (x != size - 1)
                {
                    if (!allNodes[y, x + 1].isVisited)
                    {
                        neighbors.Add(allNodes[y, x + 1]);
                    }
                }
                if (y != 0)
                {
                    if (!allNodes[y - 1, x].isVisited)
                    {
                        neighbors.Add(allNodes[y - 1, x]);
                    }
                }
                if (y != size - 1)
                {
                    if (!allNodes[y + 1, x].isVisited)
                    {
                        neighbors.Add(allNodes[y + 1, x]);
                    }
```

```
            }

            return neighbors;
        }

        Node PickRandomFromNeighbors(List<Node> neighbors)
        {
            Random rnd = new Random();
            return neighbors[rnd.Next(0, neighbors.Count)];  //can implement if statement if
0,0 doesnt work
        }

        void RemoveWall(Node node1, Node node2)
        {
            int xDif = node1.x - node2.x;
            int yDif = node1.y - node2.y;

            if (yDif == 1)  //Top
            {
                node1.walls[0] = true;
                node2.walls[2] = true;
            }
            if (xDif == -1) //Right
            {
                node1.walls[1] = true;
                node2.walls[3] = true;
            }
            if (yDif == -1) //Bottom
            {
                node1.walls[2] = true;
                node2.walls[0] = true;
            }
            if (xDif == 1)  //Left
            {
                node1.walls[3] = true;
                node2.walls[1] = true;
            }
        }

        int WallAmount(Node node)
        {
            int wallAmount = 0;
            int x = node.x;
            int y = node.y;

            foreach (var wall in node.walls)
            {
                //no need for checking if border node
                if (!wall)
                {
```

```
            wallAmount++;
        }
    }

    return wallAmount;
}

void RemoveRandomWall(Node node1)
{
    int x = node1.x;
    int y = node1.y;
    List<Node> walledNeighbors = new List<Node>();

    if (y != 0)
    {
        if (node1.walls[0] == false)
        {
            walledNeighbors.Add(allNodes[y - 1, x]);
        }
    }   //top

    if (x != size - 1)
    {
        if (node1.walls[1] == false)
        {
            walledNeighbors.Add(allNodes[y, x + 1]);
        }
    }   //right

    if (y != size - 1)
    {
        if (node1.walls[2] == false)
        {
            walledNeighbors.Add(allNodes[y + 1, x]);
        }
    }   //bottom

    if (x != 0)
    {
        if (node1.walls[3] == false)
        {
            walledNeighbors.Add(allNodes[y, x - 1]);
        }
    }   //left

    RemoveWall(node1, PickRandomFromNeighbors(walledNeighbors));
}

void Setup()
{
```

```
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            allNodes[i, j] = new Node(j, i);
        }
    }
}   //Sets up the maze board

Setup();
allNodes[0, 0].isVisited = true;
stack.Push(allNodes[0, 0]);

while (stack.Count > 0)
{
    Node current = (Node)stack.Pop();
    List<Node> unvisitedNeighbors = FindUnvisitedNeighbors(current);

    if (unvisitedNeighbors.Count > 0)
    {
        stack.Push(current);
        Node chosen = PickRandomFromNeighbors(unvisitedNeighbors);
        RemoveWall(current, chosen);
        chosen.isVisited = true;
        stack.Push(chosen);

    }

}   //Main Maze Construction

if (braidOn)
{
    foreach (var node in allNodes)
    {
        int wallAmount = WallAmount(node);
        if (wallAmount > 2)
        {
            RemoveRandomWall(node);
        }
    }
}   //Dead End Culling


bool flag = false;
for (int i = 0; i < size; i++)
{
    for (int j = 0; j < size; j++)
    {
        bool flag2 = false;
        for (int k = 0; k < 4; k++)
```

```
        {
            if (allNodes[i, j].walls[k])
            {
                flag2 = true;
            }
        }
        if (!flag2)
        {
            flag = true;
        }
    }
}   //Maze Dimensions Output
if (!flag && captionsOn)
{
    Console.WriteLine("Success");
}   //Checking if any node is isolated


foreach (var node in allNodes)
{
    node.isVisited = false;
}   //reset variables


//DIJKSTRA
Node startNode = allNodes[size / 5, size / 5];
Node endNode = allNodes[4 * size / 5, 4 * size / 5];    //other end of maze

List<Node> unvisitedNodes = new List<Node>();
for (int i = 0; i < size; i++)
{
    for (int j = 0; j < size; j++)
    {
        unvisitedNodes.Add(allNodes[i, j]);
    }
}   //fill unvisitedNodes

List<Node> FindUnvisitedReachableNeighbors(Node inputNode)
{
    List<Node> neighbors = new List<Node>();
    int x = inputNode.x;
    int y = inputNode.y;

    if (x != 0)
    {
        if (unvisitedNodes.Contains(allNodes[y, x - 1]) && inputNode.walls[3])
        {
            neighbors.Add(allNodes[y, x - 1]);
        }
    }
```

```
    if (x != size - 1)
    {
        if (unvisitedNodes.Contains(allNodes[y, x + 1]) && inputNode.walls[1])
        {
            neighbors.Add(allNodes[y, x + 1]);
        }
    }
    if (y != 0)
    {
        if (unvisitedNodes.Contains(allNodes[y - 1, x]) && inputNode.walls[0])
        {
            neighbors.Add(allNodes[y - 1, x]);
        }
    }
    if (y != size - 1)
    {
        if (unvisitedNodes.Contains(allNodes[y + 1, x]) && inputNode.walls[2])
        {
            neighbors.Add(allNodes[y + 1, x]);
        }
    }

    return neighbors;
}

Node PickCheapestUnvisitedNode()
{
    int lowestCost = int.MaxValue;
    Node lowestNode = null;
    foreach (Node node in unvisitedNodes)
    {
        if (node.dDistance < lowestCost)
        {
            lowestCost = node.dDistance;
            lowestNode = node;
        }
    }

    return lowestNode;
}

var dijkstraWatch = System.Diagnostics.Stopwatch.StartNew();
//Dijkstra start
Node currentNode = startNode;
currentNode.dDistance = 0;
while (currentNode != endNode)
{
    if (currentNode == null)
    {
        break;
```

```
            }
            List<Node> availableNeighbors =
FindUnvisitedReachableNeighbors(currentNode);
            if (availableNeighbors.Count > 0)
            {
               foreach (Node neighbor in availableNeighbors)
               {
                  int cost = currentNode.dDistance + 1;
                  if (cost < neighbor.dDistance)
                  {
                     neighbor.dDistance = cost;
                     neighbor.previous = currentNode;
                  }
               }
            }

            currentNode.isVisited = true;
            unvisitedNodes.Remove(currentNode);
            currentNode = PickCheapestUnvisitedNode();
         }  //Main Body
            //Dijkstra end
         dijkstraWatch.Stop();
         if (captionsOn) { Console.Write("Dijkstra Runtime (ms): "); }
         Console.WriteLine(dijkstraWatch.ElapsedMilliseconds);

         //Shortest path output
         currentNode = endNode;
         List<Node> shortestPath = new List<Node>();
         if (endNode.previous == null)
         {
            Console.WriteLine("No path found");
         }  //path has not been found
         else
         {
            while (currentNode != startNode)
            {
               shortestPath.Add(currentNode);
               currentNode = currentNode.previous;
            }  //Path determination
            if (captionsOn) { Console.Write("Shortest path length: "); }
            Console.WriteLine(shortestPath.Count + 1);
         }  //path has been found


         currentNode = null;
         foreach (var node in allNodes)
         {
            node.isVisited = false;
            node.dDistance = int.MaxValue;
            node.previous = null;
```

```
}   //reset variables

//A STAR
List<Node> open = new List<Node>();
List<Node> closed = new List<Node>();

int visitedNodeCount = 0;

Node LowestInOpen()
{
   int lowestFCost = size * size;
   Node cheapestNode = open[0];
   foreach (var node in open)
   {

      if (node.fCost < lowestFCost)
      {
         lowestFCost = node.fCost;
         cheapestNode = node;
      }
   }

   if (cheapestNode != null)
   {
      return cheapestNode;
   }
   return null;
}

List<Node> TraversibleNotClosedNeighbors(Node inputNode)
{
   List<Node> neighbors = new List<Node>();
   int x = inputNode.x;
   int y = inputNode.y;

   if (x != 0)
   {
      if (!closed.Contains(allNodes[y, x - 1]) && inputNode.walls[3])
      {
         neighbors.Add(allNodes[y, x - 1]);
      }
   }
   if (x != size - 1)
   {
      if (!closed.Contains(allNodes[y, x + 1]) && inputNode.walls[1])
      {
         neighbors.Add(allNodes[y, x + 1]);
      }
   }
   if (y != 0)
```

```
            {
                if (!closed.Contains(allNodes[y - 1, x]) && inputNode.walls[0])
                {
                    neighbors.Add(allNodes[y - 1, x]);
                }
            }
            if (y != size - 1)
            {
                if (!closed.Contains(allNodes[y + 1, x]) && inputNode.walls[2])
                {
                    neighbors.Add(allNodes[y + 1, x]);
                }
            }

            return neighbors;
        }

        int CalculateFCost(Node inputNode)
        {
            Node current = inputNode;
            while (current != startNode)    //calculating path to start
            {
                inputNode.gCost++;
                current = current.previous;
            }

            inputNode.hCost = (int)Math.Sqrt(Math.Pow(Math.Abs(inputNode.x -
endNode.x),2) + Math.Pow(Math.Abs(inputNode.y - endNode.y),2));

            return inputNode.gCost + inputNode.hCost;
        }

        var aStarWatch = System.Diagnostics.Stopwatch.StartNew();
        startNode.fCost = CalculateFCost(startNode);
        open.Add(startNode);
        while (currentNode != endNode)
        {
            currentNode = LowestInOpen();
            open.Remove(currentNode);
            closed.Add(currentNode);

            visitedNodeCount++;

            foreach (var neighbor in TraversibleNotClosedNeighbors(currentNode))
            {
                if (!open.Contains(neighbor))
                {
                    open.Add(neighbor);
                    neighbor.previous = currentNode;
                    neighbor.fCost = CalculateFCost(neighbor);
```

```
                } else
                {
                    if (neighbor.gCost > currentNode.gCost + 1)
                    {
                        neighbor.previous = currentNode;
                        neighbor.fCost = CalculateFCost(neighbor);
                    }
                }
            }
        }   //Main Body
        aStarWatch.Stop();
        if (captionsOn) { Console.Write("A* Runtime (ms): "); }
        Console.WriteLine(aStarWatch.ElapsedMilliseconds);

        if (captionsOn) { Console.Write("A* visited nodes: ");
            Console.WriteLine(visitedNodeCount);
        }


        currentNode = endNode;
        List<Node> shortestPath2 = new List<Node>();
        if (endNode.previous == null)
        {
            Console.WriteLine("No path found");
        }   //path has not been found
        else
        {
            while (currentNode != startNode)
            {
                shortestPath2.Add(currentNode);
                currentNode = currentNode.previous;
            }   //Path determination
            if (captionsOn) { Console.Write("Shortest path length: "); }
            Console.WriteLine(shortestPath2.Count + 1);
        }   //path has been found

        Console.WriteLine("");
    }

    mainWatch.Stop();
    Console.Write("Total Runtime: ");
    Console.WriteLine(mainWatch.Elapsed);

    Console.WriteLine("------------------------------------------------");
    Console.WriteLine("");

  }

 }
}
```

10.2 Code Ouput

```
-------------------------------------------------
Startup Runtime: 00:00:00.2343633
Size: 40

31
87
19
87

30
83
10
83

31
103
16
103

28
63
5
63

23
75
10
75

24
75
13
75

26
81
13
81

25
69
4
69

24
79
```

10
79

24
63
3
63

Total Runtime: 00:00:00.6861862

Startup Runtime: 00:00:00.2263400
Size: 80

420
137
95
137

418
137
134
137

400
135
144
135

406
165
93
165

436
151
88
151

404
135
120
135

426
151
188

151

405
139
111
139

425
163
204
163

439
165
163
165

Total Runtime: 00:00:05.9816877

Startup Runtime: 00:00:00.2247201
Size: 120

2187
237
1163
237

2140
203
625
203

2150
225
811
225

2120
225
1079
225

2122
225
899
225

2150
219
906
219

2066
213
696
213

2080
231
758
231

2173
209
529
209

2130
199
554
199

Total Runtime: 00:00:30.0746075

Startup Runtime: 00:00:00.2303321
Size: 160

8032
289
3157
289

8114
297
3164
297

7047
303
2379
303

7264
285
2845
285

8702
291
3061
291

8032
275
3162
275

8205
263
1938
263

7809
271
1686
271

7545
305
3210
305

7059
287
2467
287

Total Runtime: 00:01:46.0339349

Startup Runtime: 00:00:00.2237627
Size: 200

21733
375
7970
375

21323

335
5700
335

21519
373
11313
373

25714
365
8653
365

22220
369
9832
369

23568
383
12346
383

25193
385
12538
385

24566
391
14268
391

23402
365
9923
365

21261
325
4795
325

Total Runtime: 00:05:29.5174085

Startup Runtime: 00:00:00.2262817
Size: 240

54796
413
22692
413

54284
465
38921
465

51360
411
16138
411

51444
405
17293
405

53293
435
28051
435

48828
421
22450
421

51450
443
25290
443

48345
395
11867
395

50336
435
29957
435

55647
441
35677
441

Total Runtime: 00:12:50.3569303

Startup Runtime: 00:00:00.2231192
Size: 280

102383
491
75610
491

103014
499
64205
499

102239
495
72156
495

114291
509
69881
509

110878
491
54733
491

105957
507
80189
507

102866
473
40224
473

99297
495
64165
495

94848
517
60757
517

154838
467
70791
467

Total Runtime: 00:29:06.1506214

Startup Runtime: 00:00:00.2282776
Size: 320

186551
603
168717
603

180088
521
69717
521

182661
539
91659
539

192621
599
148384
599

181899
549
116501
549

182335

577
144374
577

179879
561
147926
561

181219
571
163058
571

211029
557
106575
557

169147
513
53501
513

Total Runtime: 00:51:01.8120408

10.3 Raw Data

Maze Size	40		startup	2343633								
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	31	30	31	28	23	24	26	25	24	24	26,6	4
Dijkstra Path	87	83	103	63	75	75	81	69	79	63	77,8	20
A* Runtime	19	10	16	5	10	13	13	4	10	3	10,3	8
A* Path	87	83	103	63	75	75	81	69	79	63	77,8	20

Maze Size	80		startup	2263400								
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	420	418	400	406	436	404	426	405	425	439	417,9	19,5
Dijkstra Path	137	137	135	165	151	135	151	139	163	165	147,8	15
A* Runtime	95	134	144	93	88	120	188	111	204	163	134	58
A* Path	137	137	135	165	151	135	151	139	163	165	147,8	15

Maze Size	120		startup	2247201								
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	2187	2140	2150	2120	2122	2150	2066	2080	2173	2130	2131,8	60,5

Dijkstra Path	237	203	225	225	225	219	213	231	209	199	218,6	19
A* Runtime	1163	625	811	1079	899	906	696	758	529	554	802	317
A* Path	237	203	225	225	225	219	213	231	209	199	218,6	19

Maze Size	160	startup	2303321	

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	8032	8114	7047	7264	8702	8032	8205	7809	7545	7059	7780,9	827,5
Dijkstra Path	289	297	303	285	291	275	263	271	305	287	286,6	21
A* Runtime	3157	3164	2379	2845	3061	3162	1938	1686	3210	2467	2706,9	762
A* Path	289	297	303	285	291	275	263	271	305	287	286,6	21

Maze Size	200	startup	2237627	

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	21733	21323	21519	25714	22220	23568	25193	24566	23402	21261	23049,9	2226,5
Dijkstra Path	375	335	373	365	369	383	385	391	365	325	366,6	33
A* Runtime	7970	5700	11313	8653	9832	12346	12538	14268	9923	4795	9733,8	4736,5
A* Path	375	335	373	365	369	383	385	391	365	325	366,6	33

Maze Size	240	startup	2262817	

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	54796	54284	51360	51444	53293	48828	51450	48345	50336	55647	51978,3	3651
Dijkstra Path	413	465	411	405	435	421	443	395	435	441	426,4	35
A* Runtime	22692	38921	16138	17293	28051	22450	25290	11867	29957	35677	24833,6	13527
A* Path	413	465	411	405	435	421	443	395	435	441	426,4	35

Maze Size	280	startup	2231192	

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	102383	103014	102239	114291	110878	105957	102866	99297	94848	154838	109061,1	29995
Dijkstra Path	491	499	495	509	491	507	473	495	517	467	494,4	25
A* Runtime	75610	64205	72156	69881	54733	80189	40224	64165	60757	70791	65271,1	19982,5
A* Path	491	499	495	509	491	507	473	495	517	467	494,4	25

Maze Size	320	startup	2282776	

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Avg	uncertainty
Dijkstra Runtime	186551	180088	182661	192621	181899	182335	179879	181219	211029	169147	184742,9	20941
Dijkstra Path	603	521	539	599	549	577	561	571	557	513	559	45
A* Runtime	168717	69717	91659	148384	116501	144374	147926	163058	106575	53501	121041,2	57608
A* Path	603	521	539	599	549	577	561	571	557	513	559	45